

**L I F C**

LABORATOIRE D'INFORMATIQUE DE L'UNIVERSITE DE FRANCHE-COMTE

EA 4269

---

## ***Specifying and Proving a Sorting Algorithm***

Elena Tushkanova — Alain Giorgetti — Olga Kouchnarenko

---

**Rapport de Recherche no RR 2009–03**

THÈME 2 –





## **Specifying and Proving a Sorting Algorithm**

Elena Tushkanova, Alain Giorgetti, Olga Kouchnarenko

Thème 2

Techniques Formelles et à Contraintes

INRIA CASSIS

**Abstract:** This work investigates the question of automaticity of algorithm proofs, through the typical example of a sorting algorithm. The first part introduces two specification languages for Java programs. In the second part one of them is used to specify a sorting algorithm by selection. The suggested specifications are enhanced until obtaining a complete solution by the current automated theorem provers. This report is a part of Elena Tushkanova's diploma project (equivalent to a master thesis) entitled "Modular Specification of Object Oriented Programs" from the Yaroslavl State University, Russia, translated from Russian into English.

**Key-words:** Verification, proof, specification, provers, Java Modeling Language, Krakatoa Modeling Language



## **Spécifier et prouver un algorithme de tri**

**Résumé :** Ce travail étudie la problématique de l'automatisation des preuves d'algorithmes, à travers l'exemple typique d'un algorithme de tri d'un tableau en langage Java. Une première partie présente deux langages de spécification pour Java. Dans la seconde partie l'un d'eux est utilisé pour spécifier un tri par sélection. La spécification est progressivement améliorée jusqu'à obtenir une version entièrement prouvée par les outils actuels de démonstration automatique. Ce rapport est la version anglaise d'une partie du mémoire de master d'Elena Tushkanova à l'université d'Etat de Yaroslavl, Russie, intitulé "Modular Specification of Object Oriented Programs".

**Mots-clés :** Vérification, preuve, spécification, démonstration automatique, Java Modeling Language, Krakatoa Modeling Language



# Specifying and Proving a Sorting Algorithm

Elena Tushkanova	Alain Giorgetti
Yaroslavl State University	LIFC – INRIA CASSIS project
Russia	University of Franche-Comté
	Besançon cedex, France

Olga Kouchnarenko  
LIFC – INRIA CASSIS project  
University of Franche-Comté  
Besançon cedex, France

## 1 Introduction

From the 16th of March to the 31st of May Elena Tushkanova has performed her diploma project of Yaroslavl State University, Russia, at the Laboratoire d'Informatique de l'Université de Franche-Comté (LIFC) under the guidance of Alain Giorgetti and Olga Kouchnarenko. This work was the first part of a six month internship supported by the INRIA CeProMi “Action de Recherche Collaborative”(ARC). This report is the English translation of a part of Elena Tushkanova’s diploma project report, entitled “Modular Specification of Object Oriented Programs”.

One of the objectives of the ARC is the specification and modular proof of properties of imperative Java or C programs. A well conceived program is developed in a modular way, that is by the structured assembly of simpler components. The objective is also to get modularity to specify and prove these modular programs.

People in this project use the Krakatoa Modeling Language (KML). It is a specification language for Java programs. This work prepares future extensions of KML language to make object oriented program specification and proof easier.

Our proposals are illustrated with the example of an algorithm to sort a Java array. We choose this algorithm because every programmers know it and its specifications and proofs it have already been proposed [6, 3, 9]. They will serve to compare approaches. We especially address a question about automaticity: can we prove this algorithm automatically?

The resulting array must satisfy two properties:

1. The output is in increasing or decreasing order.
2. The output is a permutation, or reordering, of the input.

In [3] the sorting algorithm specification is incomplete because there is just the first condition and nothing about the second condition.

In [6] the first condition is presented by the predicate (`sorted t i j`) which expresses that array `t` is sorted in increasing order between the bounds `i` and `j`. The second condition is presented by the predicate (`permut t tt`) where `t` and `tt` are permutations of each other. This paper describes many ways to define such a predicate, but the best solution is to express that the set of permutations is the smallest equivalence relation containing the transpositions, i.e. exchanges of two elements. The predicate (`exchange t tt i j`) is defined for two arrays `t` and `tt` and two indexes `i` and `j` and the predicate (`permut t tt`) is defined inductively for the following properties: reflexivity, symmetry and transitivity. This algorithm is written in the Why language. Its syntax combines functional and imperative features. It is closed to the syntax of the Caml programming language [1] and is proved with the Coq proof assistant [2].

The same algorithm is written in Java [9] and the same specification is written in KML [10]. The algorithm is proved with the Simplify prover. More details can be found in the Section 3.2.

However the idea to introduce an inductive predicate like `permut` is not natural for Java programmers. They could find it difficult to guess which axioms defining it are useful for the proof. In fact all the difficulty in the proof is hidden in this definition. So, we would like to prove the same algorithm without this predicate, by the property 2 saying that the initial array and the resulting array have the same content.

This document is organized as follows. Section 2 presents two specification languages. Section 3 proposes specifications of a sorting algorithm and discusses whether they can be proved automatically.

## 2 Specification Languages

A specification language is a formal language used in computer science during requirement analysis and system design. Most programming languages are directly executable formal languages. They are used to implement a system. Specification languages are generally not directly executed. They describe the system at a much higher level than a programming language. There are many specification languages like CASL, JML, Spec#, Z, B, etc.



In this section we describe two specification languages for the Java programming language, namely Java Modeling Language (JML) and Krakatoa Modeling Language (KML). JML is a well-known specification language. Currently, more and more tools aiming at the verification of Java programs are adopting JML as property specification language (see [4] for an overview). KML is a new specification language for Java programs. It is designed to reduce the distance between programming and proving programs. KML has significant differences with JML.

This section is organized as follows: Section 2.1 presents Java Modeling Language, Section 2.2 presents Krakatoa Modeling Language and Section 2.3 presents a graphical user interface for proving Java programs annotated with KML specifications.

## 2.1 Java Modeling Language

The Java Modeling Language (JML<sup>1</sup>) is a specification language for Java programs, using **preconditions**, **postconditions** and **invariants**. Specifications are written as Java annotation comments to the Java program after `//@ ...` or between `/*@` and `@*/`. Hence the Java program can be compiled with any Java compiler. JML is syntactically and semantically close to Java, thus making specifications more accessible to Java programmers.

We present the basic features of JML. For more details the interested reader is referred to papers from <http://www.eecs.ucf.edu/~leavens/JML/papers.shtml>.

### 2.1.1 Basic JML keywords

- A **requires** clause specifies method precondition. Any number of **requires** clauses can be included in the single specification case. Multiple **requires** clauses in a specification case mean the same as a single **requires** clause whose precondition predicate is the **conjunction** of these precondition predicates in the given **requires** clauses. For example,

```
requires P;
requires Q;
```

means the same thing as:

```
requires P && Q;
```

---

<sup>1</sup>See <http://www.jmlspecs.org>.

- An **ensures** clause specifies a normal postcondition, i.e., a property that is guaranteed to hold at the end of the method (or constructor) invocation in the case that this method (or constructor) invocation returns without throwing an exception.

**ensures** Q;

Multiple **ensures** clauses in a specification case mean the same as a single **ensures** clause whose postcondition predicate is the **conjunction** of the postcondition predicates in the given **ensures** clauses, i.e.

**ensures** P;  
**ensures** Q;

means the same as

**ensures** P && Q;

- In a specification case a **signals** clause specifies the exceptional or abnormal postcondition, i.e., the property that is guaranteed to hold at the end of a method (or constructor) invocation when this method (or constructor) invocation terminates abruptly by throwing a given exception

**signals** (E e) P;

- An **assignable** clause is used in a method contract to specify which parts of the system state may change as the result of the method execution

**assignable** A;

- In general terms, a **pure** feature has no side effects when executed. In essence **pure** only applies to methods and constructors.

**Invariants** are properties that must be maintained by all methods

**invariant** inv;

Invariants are implicitly included in all pre- and postconditions.

### 2.1.2 Basic JML expressions

The **\result** keyword can be used in **ensures** clauses of a non-void method. Its value is the value returned by the method. Its type is the return type of the method; hence it is a type error to use **\result** in a void method or in a constructor. An expression of the form **\old(Expr)** refers to the value that the expression **Expr** had in the pre-state of a method. An expression of

the form `\pre(Expr)` also refers to the value that the expression `Expr` had in the pre-state of a method. Expressions of this form may only be used in assertions that appear in the methods bodies. The quantifiers `\forall` and `\exists` are universal and existential quantifiers (respectively). The body of a universal or existential quantifier must be of boolean type. The type of a universal or existential quantified expression as a whole is boolean. The operator `==>` works only on boolean-subexpressions. The expressions on either side of this operator must be of boolean type, and the type of the result is also boolean. If `a` is false in `a ==> b`, then the expression is true and `b` is not evaluated. For more explanation look at the paper [8].

### 2.1.3 Design by contract

Design by contract (DBC) (see [7] for an overview) is a method for developing software. The principal idea behind DBC is that a class and its clients have a “contract” with each other. The client must guarantee certain conditions before calling a method defined by the class, and in return the class guarantees certain properties that will hold after the call. The contracts are defined by program code in the programming language itself, and are translated into executable code by the compiler. Thus, any violation of the contract that occurs while the program is running can be detected immediately.

A contract in software specifies both obligations and rights of clients and implementors. For example,

```
/*@ requires R;
   @ ensures E;
   @*/
```

The method precondition `R` specifies what must be true to call it. The method postcondition `E` specifies what must be true when it terminates.

## 2.2 Krakatoa Modeling Language

Krakatoa expects as input a Java source file, annotated with the Krakatoa Modeling Language. KML is largely inspired from the Java Modeling Language. As JML, KML specifications are given as annotations in the source code, in a special style of comments after `/*@ ...` or between `/*@` and `@*/`.

KML also shares many features with the ANSI/ISO C Specification Language [11].

**Method contracts** are made of a precondition and a set of behaviors. The precondition is a proposition introduced by `requires` keyword which is

supposed to hold in the pre-state of the method, i.e. when it is called. It must be checked valid by the caller.

A normal behavior has the form:

```
/*@ ...
  @ behavior b:
  @ assumes A;
  @ assigns L;
  @ ensures E;
  @*/
```

The semantics is as follows:

1. In the post-state, i.e when the method returns normally, the property  $\backslash\text{old}(A) \Rightarrow E$  holds.
2. if  $A$  holds in the pre-state then each memory location (already allocated in the pre-state) that does not belong to the set  $L$  remains allocated and its value is left unchanged in the post-state.

In  $E$ , the notation  $\backslash\text{result}$  denotes the returned value, and  $\backslash\text{old}(e)$  denotes the value of  $e$  in the pre-state.

An exceptional behavior has the form

```
/*@ ...
  @ behavior b:
  @ assumes A;
  @ assigns L;
  @ signals (Exc x) E;
  @*/
```

The semantics is similar to normal behaviors, but here properties must hold when the method returns abruptly with exception  $\text{Exc}$ .

**Loop annotations** can be given just in front of loop constructs (while, for, etc.). It has the form

```
/*@ loop_invariant I
  @ for b: loop_invariant Ib;
  @ loop_variant V ;
  @*/
```

It states that  $I$  is an inductive invariant: it must hold at loop entry and be preserved by any iteration of the loop body. The loop invariant  $Ib$  must also be an inductive invariant, but only under the assumed clause  $A$  of behavior  $b$ . The loop variant, if given, must be an expression of integer type, which must decrease at each loop iteration, and remain non-negative.

Class `invariants` are declared at the level of class members. It has the form

```
/*@ invariant id: e; @*/
```

It states that property `e` must “always” hold for the current object. The `always` is quoted here because indeed an invariant may be temporarily invalid inside method execution. Also, it does not need to be true until the object constructor has returned.

### 2.2.1 Logic functions and predicates

Unlike JML, KML **does not allow pure methods** to be used in annotations. But it permits to declare new logic functions and predicates. They must be placed at the global level, i.e. outside any class declaration, and respectively have the form

```
//@ logic m id(m1 x1, .. , mn xn) = e;  
//@ predicate id(m1 x1, .. , mn xn) = p;
```

where `e` must have type `m`, and `p` must be a proposition. The types `m` and `mi` can be either Java types or purely logic types: integer, real.

### 2.2.2 Lemmas and axiomatic blocks

**Lemmas** are additional properties that can be added, usually to give hints to provers. A lemma is declared as

```
//@ lemma id: p;
```

A **predicate** may also be defined by an **inductive definition**.

```
/*@ inductive P(x1, .. , xn) {  
  @ case c1 : p1;  
  @ ..  
  @ case cn : pn;  
  @}  
@*/
```

where each `ci` is an identifier and each `pi` is a proposition. The semantics of such a definition is that `P` is the least fixpoint of the cases, i.e. is the smallest predicate (in the sense that it is false the most often) satisfying the propositions `p1`, ..., `pn`. To ensure existence of a least fixpoint, it is required that each proposition `pi` is of the form

```
\forall y1, .. , ym,
```

```

/*@ axiomatic P {
  @ type new_type;
  @ logic new_type func1;
  @ logic integer func2(new_type v, integer k);
  @ axiom name_axiom :
  @   body_axiom;
  @ }
@*/

```

Figure 1: Axiomatic block

$$h1 ==> \dots ==> h_l ==> P(t_1, \dots, t_n)$$

where  $P$  occurs only positively in hypotheses  $h_1, \dots, h_l$ .

Instead of an explicit definition, one may introduce an **axiomatic definition** for a set of types, predicates and logic functions, which amounts to declare the expected profiles and a set of axioms (Figure 1).  $P$  is the name of axiomatic block, `new_type` is a new type needed. There are two logic functions in this example and one axiom.

Like inductive definitions, there is no syntactic conditions which would guarantee axiomatic definitions to be consistent. It is usually up to the user to ensure that the introduction of axioms does not lead to a logical inconsistency.

### 2.2.3 Construct `\at` and default logic labels

Construct `\at(e, id)` refers to the value of the expression  $e$  in the state at label  $id$ . There are four predefined logic **labels**: `Pre`, `Here`, `Old` and `Post`. `\old(e)` is in fact syntactic sugar for `\at(e, Old)`.

1. The label *Here* is visible in all statement annotations, where it refers to the state where the annotation appears; and in all contracts, where it refers to the pre-state for the requires and the assumes, variant, terminates, ... clause and the post-state for other clauses.

2. The label *Old* is visible in **assigns** and **ensures** clauses of all contracts and refers to the pre-state of this contract.

3. The label *Pre* is visible in all statement annotations, and refers to the pre-state of the function it occurs in.

4. The label *Post* is visible in **assigns** and **ensures** clauses.

The more details could be found in [10].

## 2.3 Graphical user interface

The graphical user interface **gwhy** is a tool to help the user in the process of calling the various automatic provers on the goals (in order to analyze the failures and to determine what is wrong in the programs, the specifications, or sometimes the provers). On the left side appears the list of all goals (either from goal commands in the Why files or resulting from verification conditions generation). In front of each goal, the result for each automatic prover is displayed. Provers can be launched by clicking on their names at the top of their columns. A green bullet means a proved goal; a red bullet means an unproved goal (that may be either a “don’t know” or an “invalid” answer from the prover); the scissors means a timeout from the prover (the timeout limit can be set in the bottom bar); finally, a “tools” icon means an unexpected failure in the prover call.

The top right window displays the currently selected goal. In this window, a right click on some identifier displays its location in the corresponding Why input file in the bottom right window. This section refers to [5].

## 3 Specification of a Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of an array in a certain order. The output must satisfy two properties:

1. The output is in increasing or decreasing order.
2. The output is a permutation, or reordering, of the input.

There are many sorting algorithms for arrays like Quicksort, Selection sort, Bubble sort, etc, which are defined in [12].

This section is organized as follows: Section 3.1 presents the sorting algorithm by selection in Java, Section 3.2 presents this algorithm completed with a specification in KML. In Section 3.3, 3.4 and 3.5 we try to specify the array content with a bag and to prove the sorting algorithm automatically. Section 3.6 presents steps to help the prover completing the proof.

### 3.1 Selection sort in Java

The sorting algorithm by selection is written in Java in Figure 2. There are two methods. Method **swap** just exchanges two array elements of given indexes. In method **selectionSort**, **i** and **mi** are indexes for the current element and the minimal element respectively. **mv** serves to store this minimal

```
1. class Sort {
2.     /** method swapping 2 elements */
3.     void swap(int t[], int i, int j) {
4.         int tmp = t[i];
5.         t[i] = t[j];
6.         t[j] = tmp;
7.     }
8.
9.     void selectionSort(int t[]) {
10.        int i, j;
11.        int mi, mv;
12.        for (i = 0; i < t.length - 1; i++) {
13.            mv = t[i];
14.            mi = i;
15.            for (j = i + 1; j < t.length; j++) {
16.                if (t[j] < mv) {
17.                    mi = j;
18.                    mv = t[j];
19.                }
20.            }
21.            swap(t, i, mi);
22.        }
23.    }
24.}
```

Figure 2: Selection sort in Java

element. This algorithm divides the array into two parts: the subarray of items already sorted, built up from left to right and found at the beginning, and the subarray of items remaining to be sorted, occupying the remainder of the array. The algorithm works as follows:

1. Find the minimal value in the subarray remaining to be sorted.
2. Swap it with the value in the first position of this subarray.
3. Eliminate this position from the subarray to be sorted and repeat the steps above for the new subarray remaining to be sorted.

We can test this algorithm with different array examples, but we cannot be sure that it is always correct, i.e. it satisfies properties 1 and 2 for any array. A formal specification of these two properties is the first step towards a formal proof of its correctness.



### 3.2 Sorting algorithm with a KML specification

The selection sort algorithm is realized in `Sort.java`. This file can be found in [9].

```
/*@ requires t != null &&
   @    0 <= i < t.length && 0 <= j < t.length;
   @ assigns t[i],t[j];
   @ ensures Swap{Old,Here}(t,i,j);
   @*/
void swap(int t[], int i, int j)
```

Figure 3: Specification of the method `swap`

The specification of the method `swap` is given in Figure 3. It tells that the array `t` is not null and that only two array elements may change. The postcondition says that some instance of the predicate `Swap` holds. This predicate `Swap` is reproduced in Figure 4. `Swap{L1,L2}(a,i,j)` is true if and only if the value of `a[i]` in the state at label `L2` equals to the value of `a[j]` in the state at label `L1`, the value of `a[j]` in the state at label `L2` equals to the value of `a[i]` in the state at label `L1`, and the value of `a[k]` is the same in both states, if `k` is different from `i` and `j`.

Property 1 is expressed with the predicate `Sorted`.

```
/*@ predicate Sorted{L}(int a[], integer l, integer h) =
   @   \forall integer i; l <= i < h ==>
   @     \at(a[i],L) <= \at(a[i+1],L) ;
   @*/
```

Property 2 is specified in `Sort.java` with the `Permut` inductive predicate (Figure 5) which defines four properties: reflexivity, symmetry, transitivity and swap.

```
/*@ predicate Swap{L1,L2}(int a[], integer i, integer j) =
   @   \at(a[i],L1) == \at(a[j],L2) &&
   @   \at(a[j],L1) == \at(a[i],L2) &&
   @   \forall integer k; k != i && k != j ==>
   @     \at(a[k],L1) == \at(a[k],L2);
   @*/
```

Figure 4: Predicate `Swap`

---

```

/*@ inductive Permut{L1,L2}(int a[], integer l, integer h) {
  @ case Permut_refl{L}:
  @   \forall int a[], integer l h; Permut{L,L}(a, l, h) ;
  @ case Permut_sym{L1,L2}:
  @   \forall int a[], integer l h;
  @   Permut{L1,L2}(a, l, h) ==> Permut{L2,L1}(a, l, h) ;
  @ case Permut_trans{L1,L2,L3}:
  @   \forall int a[], integer l h;
  @   Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h) ==>
  @   Permut{L1,L3}(a, l, h) ;
  @ case Permut_swap{L1,L2}:
  @   \forall int a[], integer l h i j;
  @   l <= i <= h && l <= j <= h && Swap{L1,L2}(a, i, j) ==>
  @   Permut{L1,L2}(a, l, h) ;
  @ }
@*/

```

Figure 5: Inductive predicate `Permut`

```

/*@ requires t != null;
  @ behavior sorts:
  @   ensures Sorted(t,0,t.length-1);
  @ behavior permuts:
  @   ensures Permut{Old,Here}(t,0,t.length-1);
@*/
void selectionSort(int t[]){...}

```

Figure 6: Specification of the method `selectionSort`

The specification of the method `selectionSort` is presented in Figure 6. `sorts` and `permuts` are two behavior names. For more details, refer to page 23 in [11]. `Old` means old state, `Here` means current state. For more explanations please look at page 34 in [11] or previous section. The precondition tells that array `t` is not null. The postcondition in behavior `sorts` uses the predicate `Sorted` to say that the array `t` is sorted in increasing order. The postcondition in behavior `permuts` uses the inductive predicate `Permut` to say that the resulting array is a permutation of the starting array.

Figure 7 presents the specification of the first `for` loop. The loop invariant says that `i` is non-negative at the loop entry and stay non-negative during the execution of all the iterations of the loop body. The specification of the second `for` loop is given in Figure 8. The loop invariant states that `i` is less

```

/*@ loop_invariant 0 <= i;
   @ for sorts:
   @   loop_invariant Sorted(t,0,i) &&
   @   (\forall integer k1 k2 ;
   @     0 <= k1 < i <= k2 < t.length ==> t[k1] <= t[k2]);
   @ for permuts:
   @   loop_invariant
   @   Permut{Pre,Here}(t,0,t.length-1);
   @*/

```

Figure 7: Specification of the external loop

than  $j$ , that  $mi$  is between  $i$ , included and  $t.length$ , excluded and that  $mv$  equals to  $t[mi]$ .

```

/*@ loop_invariant
   @   i < j &&
   @   i <= mi < t.length &&
   @   mv == t[mi];
   @ for sorts:
   @   loop_invariant
   @   (\forall integer k; i <= k < j ==> t[k] >= mv);
   @ for permuts:
   @   loop_invariant
   @   Permut{Pre,Here}(t,0,t.length-1);
   @*/

```

Figure 8: Specification of the internal loop

Figure 9 shows a screenshot of the graphical interface gwhy presented in Section 2 attempting to prove this KML specification. Parts of it are proved by provers like Alt-Ergo and Simplify but no prover proves it completely. At the present time the specification is not proved by Yices.

The proof results are satisfactory but the idea to define an inductive predicate like `Permut` to specify a sorting algorithm is not that natural. We investigate another approach in the remainder of this section. We try to prove the same algorithm but with a property 2 saying that the initial array and the resulting array have the same content. The algorithm and its specification are now defined in a new Java file named `SortBag.java`.

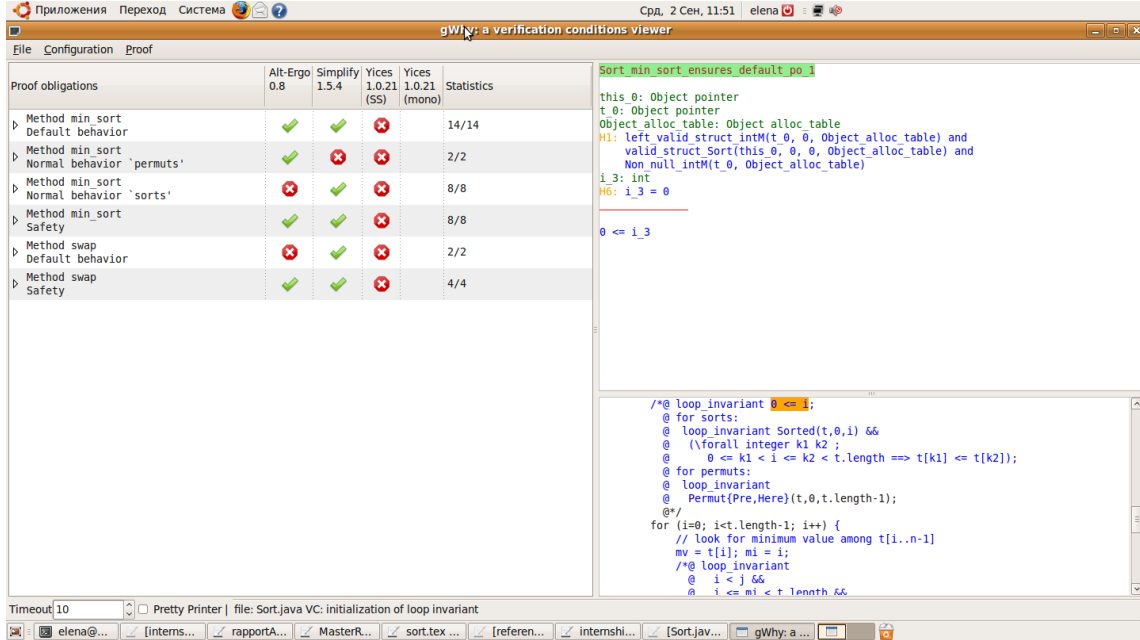


Figure 9: Proving sorting algorithm with inductive predicate

### 3.3 Bag as a model field

A bag (or multiset) is a collection without order. We want to associate to each array the bag of its elements, and to express that the output array is a permutation of the input array by writing that the corresponding bags are the same. It is a new way to prove property 2.

We describe a new type of bags by the functions from Figure 10 and the set of first-order axioms from Figure 11 that present some properties of bags. The first four axioms tell that `union` is associative, commutative and that the `empty_bag` is a neutral element for the union of bags. The next axiom tells that the number of occurrences of element `n` in bag `b` is non-negative. We add this axiom because KML does not have a type for natural integers. The subsequent four axioms characterize a function `occ` by its relation with functions `empty_bag`, `singleton` and `union`. The last axiom defines bag equality by extensionality.

The `IntArray` class is the first way we have found to link a Java array with a bag, with a model field. We declare a Java class `IntArray` with a model field

```
//@ model ibag content;
```

and implement two methods `get` and `set` as shown in Figure 12.

LIFC

```

/*@ type ibag;

// empty bag
/*@ logic ibag empty_bag();

// create_bag(n,v) is n times the value v
/*@ logic ibag create_bag(integer n, integer val);

// singleton(n)
/*@ logic ibag singleton(integer n);

// union b1 and b2
/*@ logic ibag union(ibag b1, ibag b2);

// number of occurrences of element n in bag b
/*@ logic integer occ(integer n, ibag b);

```

Figure 10: Signature for bags

The `length` behavior tells that the array length does not change. The `assignment` behavior tells that we change just one element in the array. The other elements stay unchanged. These behaviors are proved. The `content` behavior is not provable, because it is a refinement property, not a classical postcondition. Refinement is not yet supported by KML.

As depicted in Figure 13 difficulties appear when proving the method `get` (for pointer dereferencing) and the method `set` (for pointer dereferencing and `content` behavior).

Defining `content` as a model field is not the right way in KML because there is no relation between the Java array and its logical content. A solution is to remove the class `IntArray` and to use a hybrid function which returns a bag from an input array. This solution is presented in the next section.

### 3.4 Hybrid function

The `content` model field is replaced by a `content(int[] a)` hybrid function added to the signature of bags presented in Figure 10.

```

@ logic ibag content(int[] a);

```

In the previous version the `cont` behavior for method `swap` was

```

/*@ ..
@ ensures cont:

```

```

@ axiom union_assoc:
@   \forall ibag b1 b2 b3;
@     union(union(b1,b2),b3) == union(b1,union(b2,b3));
@ axiom union_comm:
@   \forall ibag b1 b2; union(b1,b2) == union(b2,b1);
@ axiom union_empty_id_left:
@   \forall ibag b; union(empty_bag(),b) == b;
@ axiom union_empty_id_right:
@   \forall ibag b; union(b,empty_bag()) == b;

@ axiom occ_non_negative:
@   \forall int n; \forall ibag b;
@     occ(n,b) >= 0;

@ axiom occ_empty:
@   \forall int n; occ(n, empty_bag()) == 0;
@ axiom occ_singleton_eq:
@   \forall int n; occ(n, singleton(n)) == 1;
@ axiom occ_singleton_neq:
@   \forall int n m; n != m ==>
@     occ(n, singleton(m)) == 0;
@ axiom occ_union:
@   \forall int n; \forall ibag b1 b2;
@     occ(n, union(b1,b2)) == occ(n, b1) + occ(n,b2);

@ axiom bag_ext:
@   \forall ibag b1 b2;
@     (\forall int n;
@       occ(n, b1) == occ(n,b2)) ==> b1 == b2;

```

Figure 11: Algebraic specification of bags

```

/*@ requires 0 <= index < a.length && a != null;
   @ assigns \nothing;
   @ ensures \result == a[index];
   @*/
public int get(int index) {...}

/*@
   @ requires 0 <= index < a.length && a != null;
   @ assigns a[index];
   @ behavior length : ensures
   @   \at(a.length,Here) == \at(a.length,Old);
   @ behavior assignment : ensures
   @   (\at(a[index],Here) == value &&
   @     (\forall integer j; 0 <= j < a.length && j != index
   @       ==> \at(a[j],Here) == \at(a[j],Old)));
   @ behavior content : ensures
   @   (\exists ibag b;
   @     \at(content,Old) ==
   @       union(b, singleton(\at(a[index],Old))) &&
   @     \at(content,Here) == union(b, singleton(value)));
   @*/
public void set(int value, int index)

```

Figure 12: Specifications of `get` and `set` methods

Proof obligations	Ak-Ergo 0.8	Alt-Ergo 0.8 [Select]	Simplify 1.5.4	Simplify 1.5.4 (Graph)	Yices 1.0.10 (SS)	Yices 1.0.10 (mono)	Statistics
Method <code>run_sort</code> Default behavior			✓				14/14
Method <code>run_sort</code> Normal behavior 'sameCont'			✓				1/1
Method <code>run_sort</code> Normal behavior 'sorts'			✓				8/8
Method <code>run_sort</code> Safety			✓				14/14
Method <code>swap</code> Default behavior			✓				1/1
Method <code>swap</code> Normal behavior 'swaps'			✓				1/1
Method <code>swap</code> Safety			✓				14/14
Method <code>getLength</code> Default behavior			✓				1/1
Method <code>getLength</code> Safety			✓				1/1
Method <code>get</code> Default behavior			✓				1/1
Method <code>get</code> Safety			X				2/3
Method <code>set</code> Normal behavior 'assignment'			✓				5/5
Method <code>set</code> Normal behavior 'content'			X				3/4
Method <code>set</code> Default behavior			✓				2/2
Method <code>set</code> Normal behavior 'length'			✓				2/2
Method <code>set</code> Safety			X				3/4
Constructor of class <code>IntArraySort</code> Safety			✓				1/1

```

IntArraySort.run_sort_ensures_default_po_1
this_5: Object pointer
t_0: Object pointer
IntArray_a: (Object, Object pointer) memory
IntArray_content: (Object, ibag) memory
Object_alloc_table: Object alloc table
H0: left_valid_struct_IntArray(t_0, 0, Object_alloc_table)
and
valid_struct_IntArraySort(this_5, 0, 0,
Object_alloc_table) and
((Non_null_Object(t_0, Object_alloc_table) and
Non_null_intMiselect(IntArray_a, t_0),
Object_alloc_table) and
offset_max(Object_alloc_table, select(IntArray_a,
t_0)) + 1 >= 1) and
same(t_0, Object_alloc_table, IntArray_content,
IntArray_a)
i_2: int
H0: i_2 = 0
0 <= i_2
.....
@ \forall int n val;
@ size(create_bag(n,val)) == n;
@ axiom size_empty :
@ size(empty_bag()) == 0;
@ axiom size_singleton :
@ \forall int n;
@ size(singleton(n)) == 1;
@ axiom size_union :
@ \forall int ibag b; \forall int n;

```

Figure 13: The attempt to prove `IntArraySort` class

```

@ \at(t.content,Old) == \at(t.content,Here);
@ ..
@*/
void swap(IntArray t, int i, int j) {...}
    and now it is

/*@ ..
@ behavior cont:
@ ensures \at(content(a),Old) == \at(content(a),Here);
@ ..
@*/
void swap(int a[], int i, int j) {...}

```

The proof obligation for this behavior is not generated by gwhy, for the same reason. There is no relation between function `content` and the Java

LIFC



heap where array `a` lives. We did not provide any axiom talking about function `content`, and we did not give a label to this function. Labels say that the `content` result depends on the heap.

The solution to this problem is to define the link between bags and arrays in axioms. It is presented in the next section.

### 3.5 Function `boundContent`

The function `boundContent` recursively defines the relation between a bag and an array with axioms. It is a more general function than function `content`. We add the following definition in the axiomatic bloc `IntBag` presented in Figures 10 and 11.

```
@ logic ibag boundContent{L1}(int[] a, integer i, integer j);
@
@ axiom emptyContent{L4}:
@   \forall int[] a; \forall integer i j;
@     (i > j ==> boundContent{L4}(a,i,j) == empty_bag());
@
@ axiom nonemptyContent{L4}:
@   \forall int[] a, integer i j;
@     i <= j ==> boundContent{L4}(a,i,j) ==
@       union(boundContent{L4}(a,i+1,j), singleton(a[i]));
```

The `cont` behavior becomes

```
/*@ ..
@ behavior cont:
@   ensures boundContent{Old}(a,0,a.length-1) ==
@           boundContent{Here}(a,0,a.length-1);
@ ..
void swap(int a[], int i, int j) {...}
```

The problem now is that there are not enough axioms to prove the behavior `cont`. The equality `==` in this behavior has the properties of reflexivity, symmetry, transitivity and congruence. But we have five functions returning an `ibag`, namely `boundContent`, `empty_bag`, `singleton`, `create_bag` and `union`, and just two axioms defining function `boundContent`: `emptyContent` and `nonemptyContent`. We could define function `boundContent` more completely, i.e. add axioms or lemmas, but there is another way. The prover should find the axiom `bag_ext`, apply it and reason with occurrences. We have observed that no prover succeeds this selection. To help provers working so we force this selection by changing the behavior to

```

/*@ ..
@ behavior cont:
@ (\forall integer i; occ(i, boundContent{Old}(a, 0, a.length-1))
@   == occ(i, boundContent{Here}(a, 0, a.length-1)));
@ ..
@*/
void swap(int a[], int i, int j) {...}

```

Finally the sorting algorithm is proved but the `cont` behavior is proved just with the prover Yices and the `content` behavior is not proved as can be seen in Figure 14.

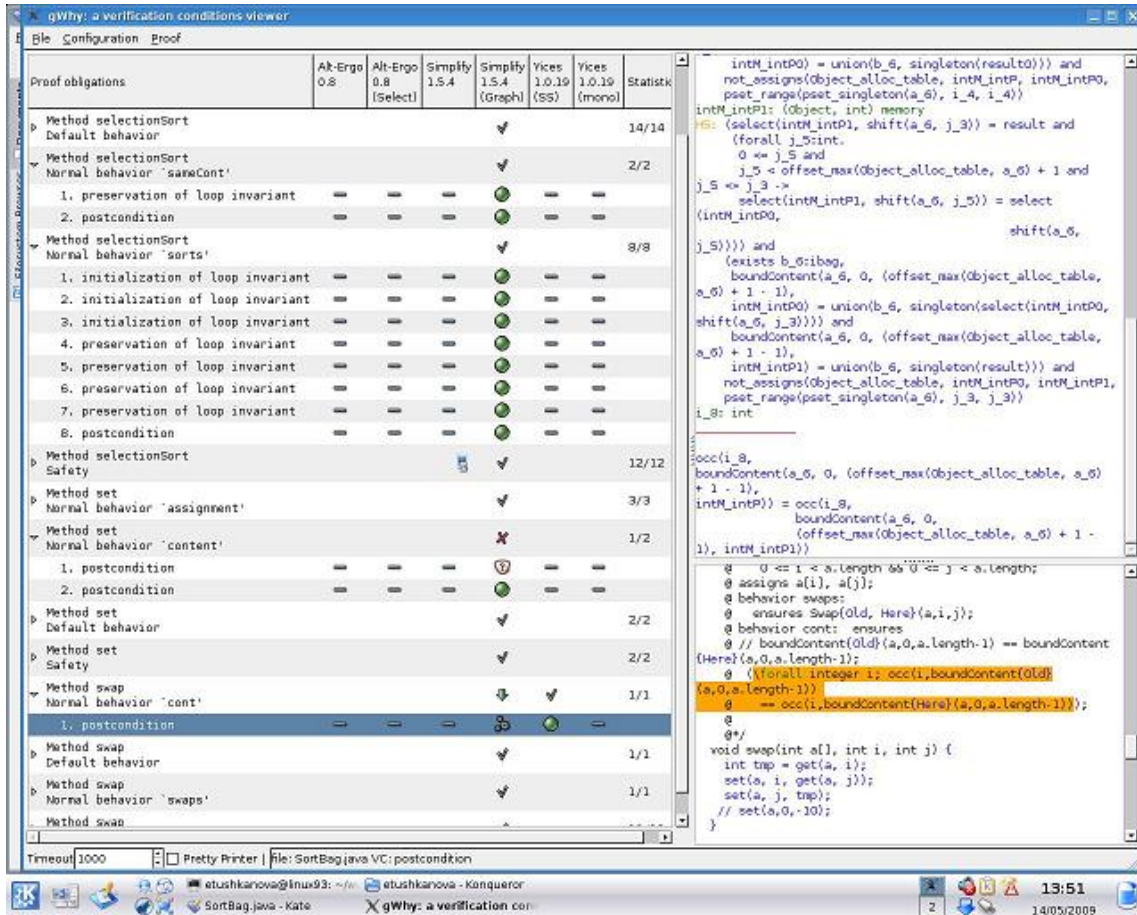


Figure 14: Attempt to prove SortBag.java

Now we would like to prove the `cont` behavior without occurrences and the `content` behavior by defining `boundContent` more completely. This experiment is presented step by step in the following two sections.

LIFC

```

@ logic ibag remove(integer n, ibag b);
@
@ axiom remove_union:
@   \forall ibag b, integer x;
@     remove(x, union(singleton(x), b)) == b;
@

```

Figure 15: Signature for bags

### 3.6 Steps for completing function boundContent

Function `boundContent` only depends on `L1`, `a`, `i` and `j`.

```

@ logic ibag boundContent{L1}(int[] a,
@                               integer i, integer j)

```

`L1` means that any heap part can be read. `a` means it depends on all the array `a`.

The **first step** is to use the KML `reads` keyword to say that `boundContent` just reads the array between `i` and `j`, it does not modify it. The `reads` keyword was a deprecated KML feature that has been re-activated by C. Marché for this purpose.

```

@ logic ibag boundContent{L1}(int[] a,
@                               integer i, integer j) reads a[i..j]

```

The **second step** is to add a function `remove` and to define it as presented in Figure 15.

The **third step** is to simplify the specification by removing the methods `set` and `get`, the function `occ` and all axioms concerned with this function.

We add the new lemma shown in Figure 16. It says that whenever the elements of an array are the same at two states, except in some position `k`, then the array content at the second state can be obtained from its content at the first state by removing the element occupying position `k` at the first state and adding the element occupying position `k` at the second state.

In method `swap` we add a new label `Middle` between label `Old` and label `Here` and three assertions to guide the provers step by step. The resulting specification is presented in Figure 17. Each assertion says what happens with `boundContent` on some step. The first assertion tells that the new content is obtained from the old content by replacing the old value of `a[i]` by the value of `a[j]` in the old content. The second assertion tells that the

```

/*@ lemma UpdateContent{L1,L2}:
@   \forall int[] a, integer i j k;
@     // update of a[k]
@     i <= k <= j &&
@     (\forall integer l;
@       i <= l <= j && k != l ==>
@       \at(a[l],L1) == \at(a[l],L2))
@   ==> boundContent{L2}(a,i,j) ==
@       union(remove(\at(a[k],L1),
@                   boundContent{L1}(a,i,j)),
@             singleton(\at(a[k],L2)));
@
@*/

```

Figure 16: New lemma

value of `a[j]` has not changed. The last assertion tells that the new content is obtained from the previous content by removing the previous value of `a[j]` and adding the value of the local variable `tmp` (which is the old value of `a[i]`).

The specification for method `selectionSort` does not change. This version of the sorting algorithm is proved with the prover `Simplify`, as shown in Figure 18.

---

```

void swap(int a[], int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    /*@ for cont: assert
       @ boundContent{Here}(a,0,a.length-1) ==
       @   union(remove(\at(a[i],Pre),
       @           boundContent{Pre}(a,0,a.length-1)),
       @   singleton(\at(a[j],Pre)));
       @*/
    /*@ for cont: assert
       @   a[j] == \at(a[j],Pre);
       @*/
    Middle: {
    a[j] = tmp;
    /*@ for cont: assert
       @ boundContent{Here}(a,0,a.length-1) ==
       @   union(remove(\at(a[j],Middle),
       @           boundContent{Middle}(a,0,a.length-1)),
       @   singleton(tmp));
       @*/
    }
}

```

Figure 17: Signature for bags

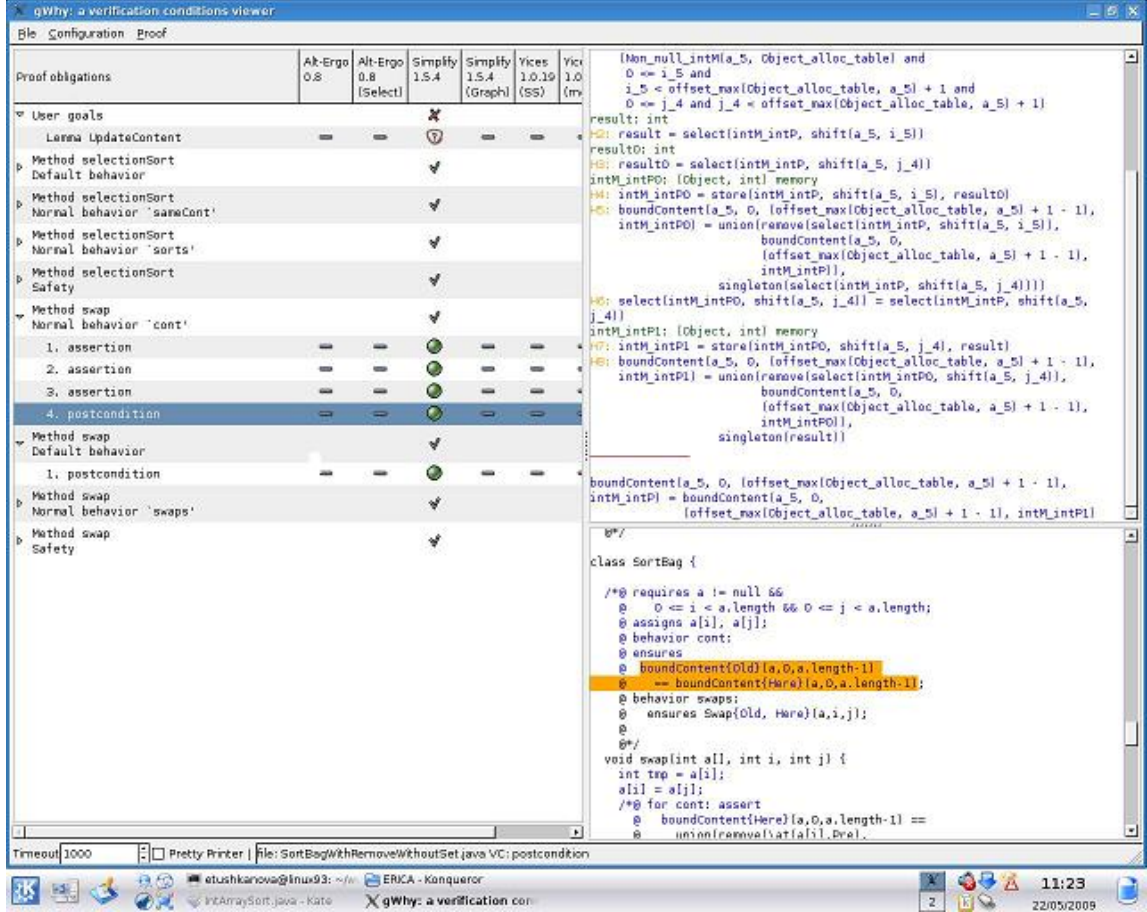


Figure 18: Proving sorting algorithm

## 4 Conclusion

We have described two specification languages for the Java programming language, namely JML and KML. To the question “Can we prove a sorting algorithm automatically?” our answer is “Yes”. We have proved a sorting algorithm by selection by using a **hybrid function** which takes an **array** as a parameter and returns a **bag**. A bag is a collection without order. Given an array, this function returns the bag of its elements. We have expressed that the output array is a permutation of the input array by writing that the corresponding bags are the same.

One consequence of our work is that the **read** keyword has been activated again in KML. This keyword is used to say that the **boundContent** reads the corresponding part of the array. This supported feature helps the provers.

The work in [6] suggests to define an inductive predicate to axiomatize the property that the output array is a permutation of the input array. Specifying with bags is more natural for Java engineers. But this new way of specifying a sorting algorithm leads to some difficulties. For instance, the `swap` method has three lines of Java code which are exchanging two elements in an array and for each line we had to write a long assertion. Each assertion specifies what happens with the array content at the corresponding Java code line. These three assertions are required by provers to succeed their proofs. Now we would like to investigate an intermediary solution, namely defining a permutation datatype. It is less familiar than bags for Java engineers, but more familiar than inductive predicates. Another direction of future work is to sort polymorphic arrays. KML does not support polymorphism yet, so we plan to implement this new feature.

## 5 Acknowledgments

We would like to thank Claude Marché for helpful discussions and suggestions.

## References

- [1] The Caml language. Available at <http://caml.inria.fr/>.
- [2] The Coq Proof Assistant. Available at <http://coq.inria.fr/>.
- [3] Z. Manna A. R. Bradley and H. B. Simpa. What's Decidable About Arrays? Available at <http://theory.stanford.edu/~arbrad/papers/arrays.pdf>, 2006.
- [4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An Overview of JML Tools and Applications. In *FMICS 03*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.
- [5] J.-C. Filliâtre. The WHY verification tool. Tutorial and Reference Manual. Available from <http://why.lri.fr/manual/manual001.html>, 2009.
- [6] J.-C. Filliâtre and N. Magaud. Certification of Sorting Algorithms in the System Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

- [7] G. T. Leavens and Y. Cheon. Design by Contract with JML. Available from <http://www.jmlspecs.org>, 2006.
- [8] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. JML Reference Manual. Department of Computer Science, Iowa State University. Available from <http://www.jmlspecs.org>, 2002.
- [9] C. Marché. Winter School: instruction for lab session/Krakatoa tool. Available at <http://krakatoa.lri.fr/ws/>.
- [10] C. Marché. The Krakatoa tool for Deductive Verification of Java Programs. Available from <http://krakatoa.lri.fr/ws/>, 2009.
- [11] J.-C. Filliâtre P. Baudin and C. Marché. ACSL: ANSI/ISO C Specification Language. Available from <http://frama-c.cea.fr/acsl.html>, 2008.
- [12] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, 1988.





Laboratoire d'Informatique de l'université de Franche-Comté  
UFR Sciences et Techniques, 16, route de Gray - 25030 Besançon Cedex (France)

LIFC - Antenne de Belfort : IUT Belfort-Montbéliard, rue Engel Gros, BP 527 - 90016 Belfort Cedex (France)  
LIFC - Antenne de Montbéliard : UFR STGI, Pôle universitaire du Pays de Montbéliard - 25200 Montbéliard Cedex (France)

---

<http://lifc.univ-fcomte.fr>